Adaptive Programming with Claude Code

A Plain-English Explanation
of a New Way to Build Software

Originally written by A. F. Urakhchin
This high school edition: May 2026

If you can program a little — even just a little — you can understand what this is about. The ideas are simpler than they sound when other people describe them.

1. What Is This Really?

1.1 The Question We're Trying to Answer

Imagine you describe a piece of software to a friend in plain English — "I want a program that reads a JSON file, picks out all the parts that match a certain pattern, and saves them somewhere else." Your friend writes the program. You run it. It works. But here's the question: how do you know it really works? Not just "it didn't crash" — how do you know it does what you actually meant?

For most software, the answer is: you don't, not really. You run some tests, you eyeball the output, you trust your friend was careful. If your friend made a subtle mistake — a bug that only shows up on weird inputs you didn't think to test — you might never find out.

This document is about a way of building software where you can actually know. Not in some loose, intuitive sense. In a strict sense — you can write a small checking program that says yes or no, this is correct or it isn't, and the checking program's answer is final. The friend in this story isn't a person; it's an AI (specifically, Anthropic's Claude Code). But the AI's work isn't trusted because the AI is smart. It's trusted because the checking program agrees.

That combination — describe-in-English, AI-writes-code, mechanical-checker-decides — is what "Adaptive Programming" means. The rest of this document explains why it works, what it needs, and why it's a big deal.

1.2 Why This Isn't Just "AI Coding Tools"

You may have heard of GitHub Copilot, Cursor, or other AI coding tools. They suggest code; you accept or reject. They're useful — most developers use them every day. But they're not what this document is about, and the difference matters.

With those tools, the human is still the judge. You read what the AI wrote, you decide if it looks right, you press accept. If you press accept on something subtly wrong, the wrongness goes into your codebase. The AI is helping you write code faster, but the question "is this correct" still has the same answer it always did: it's correct if a human said it was correct.

Adaptive Programming is different in one specific way: a checking program, not a human, decides whether the AI's work is correct. The checking program is not part of the AI. It exists separately, was built ahead of time, and works the same way regardless of what the AI produces. If the AI's output passes the check, the work is done. If it doesn't, the AI tries again. The human writes the description in English; the AI writes the code; the checker decides. Three separate roles, three separate concerns.

This is qualitatively different from "AI writes code, human approves." It's a closed loop, where "closed" means there's an actual answer to "are we done" that doesn't require human judgment in the moment.

2. The Four Ingredients

2.1 The Four Things You Need

To do Adaptive Programming, you need four things working together. Each one is necessary. Take any one of them away and the whole thing falls apart.

First: a specification. This is just a description in English of what you want. Not formal, not mathematical — actual English, with maybe some structural hints ("I want a project with three folders, one for C++, one for Python, one for JavaScript"). You write this. It's the input.

Second: an AI coding agent. Today, this is Claude Code (or one of a handful of similar tools). The agent reads your specification, figures out what to build, writes the code, runs it, debugs problems, and iterates. The agent is the worker.

Third: a substrate. This is the unusual ingredient. A substrate is a foundation of pre-built code and conventions that the AI works against — not the user's project, but a stable base underneath it. The substrate provides operations the AI can use without having to invent them. In the case we're discussing, the substrate is something called CCS (we'll get to what that is in a moment). The substrate is the foundation.

Fourth: an oracle. This is a checking program. It looks at what the AI produced and says yes-correct or no-not-correct, in a way that's mechanical — meaning a computer can run it and get a definite answer, not "it seems okay." The oracle is the judge.

These four — specification, agent, substrate, oracle — are the whole game. Everything else is detail.

2.2 A Concrete Example

Here's a real example, simplified. Suppose your specification says: "I want a small program in three languages — C++, Python, and JavaScript — that can read a JSON file, understand its structure, and rewrite it back to exactly the same bytes it started with."

That last part — "exactly the same bytes" — is what makes the oracle possible. Compare the program's output against the input, byte by byte. If they match exactly, the program works. If they don't, it doesn't. There's no judgment call. A computer can check this in milliseconds.

Now imagine the AI builds the three programs from your English description. C++ version, Python version, JavaScript version. Each one reads the JSON, processes it, writes it back. You run the oracle: for each version, did the output match the input byte-for-byte? Yes or no. If yes, the AI is done. If no, the AI tries again — looks at what didn't match, figures out why, fixes the code.

This actually happened. In May 2026, the project that demonstrated Adaptive Programming used exactly this kind of example. The AI produced about 2,500 lines of code across the three languages, in one working session. All the tests passed. There was no human "approving" anything; the byte-by-byte comparison either matched or it didn't.

The point isn't the size of the project — it's small. The point is that the AI produced working multi-language code from a verbal description, and you can know for certain it works because the comparison-check is mechanical.

3. Why the Substrate Matters So Much

3.1 The Substrate Does the Hard Part

Of the four ingredients, the substrate is the one that most needs explaining, because it's the least obvious. Why can't you just have a description, an AI, and a checker? Why do you need a foundation underneath?

Here's why. Suppose you ask the AI: "write me a JSON parser in C++." The AI does it. Now suppose you ask: "write me a JSON parser in Python that produces the same output as the C++ one." The AI does that too. But the two parsers are written separately. They might agree on most inputs and disagree on some weird ones — different handling of whitespace, different number formats, different edge cases. You wouldn't know unless you found the disagreement.

Now suppose you have a substrate that already has a working JSON parser — one that's been built and tested over years. And it has a canonical way to represent parsed JSON as bytes, the same bytes regardless of which language is doing the parsing. And it has a way to take those bytes and turn them back into text that's identical to the original. The substrate gives you all this for free, before the AI starts working.

Now when you ask the AI to write parsers in three languages, the AI doesn't write three different parsers. It writes three thin wrappers around the substrate's existing parser. Each wrapper produces the same canonical bytes. The checker compares those bytes. Of course they match — they came from the same underlying parser. The AI's job was much simpler than it looked: not "write a JSON parser three times," but "wrap an existing parser in three language-specific shells."

This is the substrate's gift. It does the hard part — the part where mistakes would normally creep in. The AI does the easy part — adapting the substrate to a specific language and specific user requirements. The checker can be definitive because there's a canonical reference (the substrate's own output) that the AI's output must match.

3.2 What CCS Actually Is

CCS stands for "Compiler Compiler System." It's a substrate that has been in development for years, with a US patent (number 8,464,232) protecting the methods it uses. What does it actually do? It's a compiler-compiler — meaning, you describe a language by writing down its grammar (the rules of what's valid in the language), and CCS produces a working compiler for that language. The compiler can read text in the language, turn it into a binary format, and then turn that binary back into text that's identical to what it read.

That last part — "turn the binary back into text that's identical to what it read" — is the key property. It's called "round-trip closure." The text-to-binary-to-text cycle returns you to your starting point, exactly. This is the property that makes mechanical checking possible: when the AI writes code that uses CCS, the AI's code's output can be compared to CCS's own canonical output, byte by byte, and the comparison is decisive.

CCS isn't the only substrate that exists. There are others — Protocol Buffers, Cap'n Proto, ANTLR — that share some properties with CCS. But none of them have all five properties that Adaptive Programming

requires. CCS is, as far as anyone can tell as of May 2026, the only production system that has the full combination. This is why CCS is the substrate that makes Adaptive Programming possible today, and it's also why the patent matters commercially — the substrate is hard to replicate.

3.3 The Five Properties That Matter

The five properties a substrate needs to support Adaptive Programming, stated in plain English:

Property 1 — the substrate has to actually do useful things. It can't just be a specification; it has to be working code that does meaningful work. CCS has cppcc, the compiler-compiler tool. That's working code.

Property 2 — there has to be a mechanical way to decide if the AI's output is correct. CCS has the round-trip property: text becomes binary becomes text, and the round-trip is byte-identical. That's the checking criterion.

Property 3 — if you're producing code in multiple languages, the substrate has to provide essentially the same operations in each language, in the same shape. CCS provides what it calls SCB APIs (Syntax-Controlled Binary APIs) in C++, Python, and JavaScript — three different syntaxes, the same operations, the same results.

Property 4 — there has to be a canonical form that all the languages agree on. CCS has its binary format (called .bfgr files) — bytes that are the same regardless of which language is working with them.

Property 5 — the substrate has to enforce round-trip closure throughout. CCS's whole architecture is built around this. It's not a feature added on top; it's the foundation.

Put these five properties together, and the AI's work becomes mechanically checkable. Take any of them away, and the work becomes ambiguous in ways the checker can't resolve.

4. Why This Is a Big Deal

4.1 Three Levels of AI-Assisted Programming

To see why Adaptive Programming matters, it helps to know where it sits compared to other ways AI helps with software.

Level 1: AI suggests code, human approves. This is GitHub Copilot, Cursor's basic mode, most of what "AI coding" means today. The AI helps you type faster. You decide if the suggestions are right. Used by millions of developers. Useful. Not what we're talking about.

Level 2: AI does longer-horizon work against a codebase. This is more recent — Devin, Cursor's agent mode, Claude Code in its default use. The AI takes a task that would have taken you an afternoon and tries to complete it autonomously. It writes tests, runs them, debugs. You read the result and decide if it's done. Still human-judged at the end.

Level 3: AI does autonomous work against a substrate with a mechanical oracle. This is Adaptive Programming. The substrate's checker decides if the work is done, not the human. This is qualitatively different from Level 2: the human doesn't read code and decide; the oracle reads bytes and decides.

Above Level 3 there are theoretical levels involving formal mathematical specifications and provable correctness — but those are different problems, in different territory. Adaptive Programming as we've defined it is Level 3.

Most agentic AI tools are at Level 1 or Level 2 as of May 2026. Level 3 is rare. CCS plus Claude Code is, as far as anyone can tell, the most clearly-demonstrated Level 3 system in existence.

4.2 What's New About This

Three things make Adaptive Programming new, and worth paying attention to.

First: the human doesn't have to write formal specifications. There's a whole field of computer science ("formal methods") that achieves mechanical correctness, but only at the cost of requiring the user to write specifications in a formal language — proofs, logical contracts, type signatures with mathematical precision. That's hard, and most software developers don't do it. Adaptive Programming achieves mechanical correctness while letting the user write specifications in plain English. The mechanical part comes from the substrate's oracle, not from the user's spec.

Second: it fuses three traditions that have been separate. There's a long tradition of building software from models or schemas (called "Model-Driven Engineering"). There's a separate tradition of building compilers from grammars ("compiler-compiler systems"). And there's the new field of AI-driven coding. CCS plus Adaptive Programming fuses all three. No single system before has done all three together with the mechanical-oracle property.

Third: the work scales. The demonstration project (ccs_scb, May 2026) was small — 2,500 lines, single session. But the same methodology is now being applied to build a much larger system — the CCT Repository, a database-like thing that took several weeks of step-by-step work. That work succeeded. The

methodology doesn't break when projects get bigger; it adapts. (The bigger projects develop additional practices, called operational disciplines, that the small worked example didn't need. But the core mechanism — verbal spec, AI work, substrate-supplied oracle — stays the same.)

4.3 Why Most AI Coding Tools Can't Do This

It's tempting to think: if Claude Code can do Adaptive Programming, then any capable AI coding tool should be able to do it. The substrate is the issue. Most users of Cursor, Devin, Aider, or other tools are working against their own codebases — codebases that don't have a substrate underneath. There's no canonical reference to compare the AI's output against. There's no oracle to decide. The AI does great work, but the work has to be human-judged.

If those users could install CCS and use it as their substrate, they would also be doing Adaptive Programming. The methodology isn't locked to Claude Code. The substrate is what's special. This is why the people building CCS plan to make it available to other AI tools as well, through something called an "MCP server" (a standard protocol AI tools use to access external capabilities). Once CCS is accessible this way, any MCP-supporting AI tool can do Adaptive Programming work.

Put differently: the substrate is the moat. The agent is replaceable. Claude Code is the agent in the demonstrations because Claude Code is the agent the people building CCS use. Other agents could in principle do the same work against the same substrate. The substrate, on the other hand, is hard to replace — building one with all five properties took decades of work and is protected by a patent.

5. What This Can and Can't Do

5.1 What Adaptive Programming Is Good For

Adaptive Programming works well for certain kinds of software problems. Specifically, the kinds where:

The work involves data formats, protocols, or languages that the substrate can describe. CCS is a compiler-compiler, so it shines on anything involving structured data with well-defined rules — JSON files, configuration languages, message protocols, custom domain-specific languages.

The correctness criterion can be expressed as a round-trip. "The output should be byte-identical to the input after processing" is a natural fit. "The output should match this canonical reference" is a natural fit.

Multiple languages are involved. The substrate's multi-language story means you write the description once, and you get implementations in C++, Python, and JavaScript that are guaranteed to agree on the wire format. This is hugely valuable when teams use different languages but need their code to interoperate.

The scope is bounded. Adaptive Programming works best on problems where the substrate's leverage is real — where most of the hard work is done by the substrate, and the AI's job is to adapt the substrate to specific user requirements. Problems that are mostly substrate-aligned are good; problems that mostly require new design outside the substrate are not.

5.2 What Adaptive Programming Is Not Good For

To be clear-eyed about it: Adaptive Programming is not a universal answer. It's a powerful approach for a particular class of problems, not all problems.

It's not good for problems the substrate can't represent. If your problem is to write a real-time graphics renderer for a video game, CCS doesn't help — there's no canonical-bytes property for a rendered frame, no round-trip oracle, no substrate leverage. The methodology has nothing to grip on.

It's not good for problems with fuzzy or aesthetic correctness criteria. "Make the user interface look nice" doesn't have a mechanical oracle. "Make the code well-organized" doesn't have a mechanical oracle. These are real, important properties of software, but they're not decidable by a checker.

It's not good when the user needs informal specs to mean what the user intends them to mean. The substrate's oracle decides whether the output matches the substrate's references. It doesn't decide whether the references match what the user actually wanted. The user has to verify intent independently. The methodology guarantees correctness against the oracle, not correctness against intent.

These limits aren't failures of the methodology; they're the methodology's natural scope. Knowing what it's for is part of using it well.

5.3 Why It Still Matters Even With These Limits

Even with the limits, Adaptive Programming covers a significant chunk of real software work. Data interchange formats. Protocol implementations. Parser and compiler tooling. Multi-language libraries.

Configuration languages. Domain-specific languages. Schema-driven systems. All of these are mainstream software engineering needs, and all of them are within the methodology's natural scope.

More importantly: in the parts of software where it works, it works very well. The 2,500-line demonstration produced correct code on first or second try, across three languages, in a single working session. Traditional methods would have taken a developer days. The speed-up isn't 2× or 3×; it's more like 10× or 20× for the right kind of problem. And the correctness is stronger than traditional methods deliver, because the oracle is mechanical rather than human-judged.

So while it's not universal, where it applies, it's transformative. And the set of problems it applies to is large enough that finding out it's available is real news.

6. What Might Come Next

6.1 Where the Work Is Going

As of May 2026, Adaptive Programming has been demonstrated on a small worked example (ccs_scb) and is being used to build a larger piece of infrastructure (the CCT Repository). The next steps in the publicly-visible plan are: finish building the Repository through Adaptive Programming; build a network transport layer the same way; build a graphical tool for managing CCT data; eventually add an AI-driven layer on top to assist users. Together, these projects form what's called the CCT bootstrap — the work of building CCT's own infrastructure using its own methodology.

In parallel, a commercial product is being designed: an MCP server (Model Context Protocol server) that makes CCT substrate capabilities available to any AI coding tool that supports MCP. This is the first commercial CCT product, targeting customers who want to use Adaptive Programming with whatever AI agent they're already using.

6.2 What Could Go Wrong

Honest version: the bet that Adaptive Programming scales beyond worked examples to production infrastructure is not yet fully proven. The CCT Repository work in progress is the first real test. If it succeeds, Adaptive Programming becomes a defensible methodology with strong evidence. If it doesn't — if the bigger work surfaces issues that small examples don't — the methodology will need refinement, possibly substantial.

The smart bet is that it scales, because the substrate's properties are general and the AI's capabilities are improving rapidly. But the bet hasn't paid off yet. As of May 2026, the project is genuinely in the middle of finding out.

6.3 What This Means for You

If you're a high school student reading this, here's what's worth taking away:

First, AI-assisted coding is going to be a much bigger deal than it currently is. Today's tools are at Level 1 or Level 2. Level 3 — what Adaptive Programming represents — is qualitatively different. Tools that achieve it will reshape how software is built in fields where they apply. Knowing this gives you a head-start on understanding where the field is going.

Second, the methodology depends on infrastructure that takes decades to build. CCS is a multi-decade project. Substrates with the right properties are hard to construct. This means there's commercial value in being one of the people who can build such substrates — and academic value in understanding the theoretical properties they need.

Third, the specific approach is more interesting than it sounds. "AI writes code, human approves" is what most people imagine when they hear "AI coding." "Human writes English, AI writes code, mechanical checker decides" is a fundamentally different design — and it produces stronger correctness with less

human effort. If you're going into software engineering, this is the kind of architectural pattern worth understanding deeply, because it's likely to appear in more places as the underlying technologies mature.

Fourth, and most concretely: if you want to play with these ideas yourself, you can. The AI tools are publicly available. Building your own substrate is hard (it's why CCS took decades), but you can experiment with the methodology against simpler oracle properties — testing equality, checksum matching, structural comparison — on problems where you can construct your own canonical references. You won't have CCS's leverage, but you can see the basic mechanism work. That's how you'd learn what's really going on, beyond what any document can tell you.

This is genuinely new territory. The big picture is still being mapped. The people doing the mapping right now are people like the engineer whose work this document explains. In ten years, when this is everywhere, you'll have known about it from near the beginning.

That's worth something.